DOI: 10.58168/PPSPM2025_362-371
УДК 519.713.1

АРХИТЕКТУРНАЯ ЭВОЛЮЦИЯ И ТЕХНОЛОГИЧЕСКИЕ РЕШЕНИЯ ФРОНТЕНД-ПРИЛОЖЕНИЯ «АТОМ.ОКО»

Е.А. Кирин, И.Р. Евчук

ФГБОУ ВО «Воронежский государственный лесотехнический университет имени Г.Ф. Морозова»

Аннотация: статья посвящена архитектурной эволюции и технологическим решениям фронтенд-приложения системы анализа и классификации документов «Атом.ОКО». Описаны основные этапы перехода от модульной структуры к Feature-Sliced Design (FSD), применение React 18, MobX и TypeScript для построения масштабируемой бизнес-логики. Рассмотрены решения по визуализации интерактивных графов с использованием React Flow, управление позиционной разметкой документов через Paper.js, а также оптимизация интерфейса и сборки при переходе с Webpack на Vite. Особое внимание уделено lazy loading компонентов, организации CI/CD, контейнеризации с Docker, обеспечению качества кода и тестированию с помощью Jest, React Testing Library и BackstopJS. Подчёркивается важность системного подхода к архитектуре, обеспечивающего масштабируемость, предсказуемость и удобство сопровождения фронтенд-приложения.

Ключевые слова: фронтенд, React, MobX, TypeScript, Feature-Sliced Design, React Flow, Paper.js, Ant Design, Vite, lazy loading, CI/CD, Docker, тестирование, Jest, React Testing Library, BackstopJS, визуальная бизнес-логика, OCR.

ARCHITECTURAL EVOLUTION AND TECHNOLOGICAL SOLUTIONS OF A FRONTEND APPLICATION «ATOM.OKO»

E.A. Kirin, Evchuk I.R.

[1]Voronezh State University of Forestry and Technologies named after G.F. Morozov

Abstract: The article is devoted to the architectural evolution and technological solutions of the frontend application of the Atom.OKO document analysis and classification system. The main stages of the transition from a modular structure to Feature-Sliced Design (FSD), the use of React

18, MobX and TypeScript for building scalable business logic are described. Solutions for visualization of interactive graphs using React Flow, control of positional markup of documents through Paper are considered.js, as well as optimizing the interface and build when switching from Webpack to Vite. Special attention is paid to lazy loading of components, CI/CD organization, containerization with Docker, code quality assurance and testing using Jest, React Testing Library and BackstopJS. The importance of a systematic approach to architecture that ensures scalability, predictability, and maintainability of a frontend application is emphasized.

Keywords: frontend, React, MobX, TypeScript, Feature-Sliced Design, React Flow, Paper.js, Ant Design, Vite, lazy loading, CI/CD, Docker, testing, Jest, React Testing Library, BackstopJS, visual Business logic, OCR.

Разработка клиентской части системы анализа и классификации документов «Атом.ОКО» началась с относительно простой модульной архитектуры, привычной для React-приложений среднего масштаба. Кодовая база была организована по техническому признаку — в отдельных директориях находились компоненты, сторы, API, утилиты. На старте это обеспечивало быстрый темп разработки, но с ростом проекта подход стал проявлять ограниченность. Количество сценариев и бизнес-логики росло, модули начали обрастать несвойственными зависимостями, а границы ответственности размывались. Появились ситуации, когда компонент интерфейса напрямую взаимодействовал с состоянием стороннего бизнес-процесса, что нарушало изоляцию и усложняло рефакторинг. Возникла необходимость в архитектурной перестройке, способной обеспечить масштабируемость и прозрачность кода в долгосрочной перспективе.

Ответом на этот вызов стало внедрение методологии Feature-Sliced Design (FSD), предложившей структурировать проект в соответствии с бизнес-логикой, а не типом файлов. Эта методология ввела четкое разделение слоев — shared, entities, features, widgets и pages, где зависимости направлены строго от общего к частному. Благодаря FSD удалось локализовать бизнес-сущности, разделить зоны ответственности и устранить неявные связи.

Архитектурная структура приобрела читаемость и предсказуемость, что значительно облегчило онбординг новых разработчиков. Для обеспечения дисциплины в проект был интегрирован линтер архитектуры (steiger), запускаемый командой npm run lint:fsd, который автоматически отслеживает нарушения архитектурных правил. Таким образом, архитектурная чистота была закреплена на уровне процессов сборки и ревью, предотвращая деградацию структуры с течением времени.

Базой фронтенд-приложения стал React 18. Этот выбор продиктован зрелостью фреймворка, широтой экосистемы и декларативным подходом к построению интерфейсов. Использование функциональных компонентов и хуков позволило создавать переиспользуемую, модульную бизнес-логику, изолируя состояние и побочные эффекты. React оказался особенно эффективен в задачах, связанных с динамическим отображением данных — таблиц, аннотаций, форм и графов.

Для управления состоянием была выбрана библиотека MobX, предложившая реактивную и при этом простую в освоении модель. В отличие от Redux, который требует иммутабельных редьюсеров и строгого описания всех действий, MobX позволяет работать с мутабельными объектами, автоматически отслеживая зависимости. Это оказалось решающим преимуществом в интерфейсах с высокой частотой обновлений данных — например, при аннотировании документов или редактировании шаблонов. Благодаря mobx-react-lite рендеринг компонентов стал избирательным: перерисовываются только те, чьи наблюдаемые поля действительно изменились. Такой подход обеспечил отзывчивость интерфейса и уменьшил накладные расходы на вычисления. Архитектурно сторы оформлены в виде классов (например, DocTypeModel, UserModel), где методы действий объявлены как экшены (@action), что делает бизнес-логику читаемой и разделённой.

Особое значение во фронтенд-инфраструктуре имеет TypeScript. В работе над сложной предметной областью, типизация является не только инструментом безопасности, но и занимает важную роль в документации. Типы могут заниматься описанием как контрактов между различными слоями приложения, так и структур запросов и ответов API между фронтендом и бэкендом. Это значительно уменьшает количество ошибок, которые возникают из-за несоответствия данных, и позволяет упростить масштабирование системы. Кроме того, типизация поддерживает стандартизацию взаимодействия с внешними сервисами, в том числе OCR и Keycloak, что, в свою очередь, обеспечивает предсказуемость и стабильность API-интерфейсов.

В рамках развития системы «Атом.ОКО» возникла необходимость наглядно представлять сложные бизнес-процессы. Решением стала разработка интерактивного конструктора, с помощью которого пользователи имели возможность самостоятельно проектировать сценарии работы с данными. Такой подход позволит не только визуально представить ход выполнения

требуемых действий, но и полностью избежать необходимости писать или править код вручную. Предлагаемые сценарии должны поддерживать различные операции над данными, например, сравнение, арифметические вычисления, преобразования типов и сохранение результатов. Требовалось обеспечить удобство и интуитивную ясность взаимодействия пользователя с системой, а также гарантировать логическую корректность построенных процессов.

В рамках исследования были проанализированы различные подходы: создание собственного редактора на canvas-основе, использование графовых библиотек общего назначения D3.js и специализированных решений для построения блок-схем. Результаты анализа показали, что библиотека React Flow (@xyflow/react) наилучшим образом сочетает гибкость, производительность и возможность глубокой интеграции с React-экосистемой. Она предоставила готовый инструментарий для построения интерактивных графов, включающий drag-and-drop, множественные входные и выходные хендлы, динамическое изменение структуры узлов и встроенную систему событий. Это позволило реализовать редактирование графа без необходимости разрабатывать собственный движок рендеринга связей и маршрутизации сигналов.

В отличие от типичных сценариев использования React Flow, где акцент делается на визуализацию данных, библиотека стала ядром редактора бизнес-логики, тесно связанного с состоянием приложения и серверной моделью данных. Графовое представление (узлы и связи) синхронизируется с централизованным состоянием, управляемым через MobX, что обеспечивает реактивное обновление интерфейса и согласованность данных. Каждый узел интерпретируется как атомарная операция — сравнение, преобразование или сохранение переменной, — а связи между ними формируют ориентированный ациклический граф вычислений. Такая модель обеспечивает детерминированность выполнения сценариев и возможность их сериализации в формат, понятный серверной части.

Особое внимание было уделено валидации графа, обеспечивающей логическую корректность собранных сценариев. Разработанная многоуровневая система проверок включает контроль структурной целостности (отсутствие циклов и несвязанных компонентов), ограничение на единственный узел сохранения переменной, верификацию типов данных между операциями и обнаружение «висячих» выходов. Эти проверки выполняются динамически при

каждом изменении графа, что объединяет визуальное взаимодействие с мгновенным контролем правильности.

В случае нарушения правил система отображает ошибки в контексте соответствующих элементов, направляя пользователя к исправлению некорректных связей.

Исследовательский интерес данного решения заключается в демонстрации того, как визуальные средства моделирования могут повысить доступность системы для пользователей, не обладающих технической подготовкой. Возможность собирать бизнес-логику в виде графа снижает порог входа, делает процесс построения сценариев более интуитивным и способствует прозрачному пониманию вычислительных процессов. Таким образом, использование React Flow в сочетании с MobX позволило объединить интерактивность пользовательского интерфейса с формальной строгостью бизнес-логики, создав устойчивую основу для визуального программирования внутри веб-приложения.

Для задач аннотирования и интерактивного выделения областей документов применялась библиотека Paper.js – мощное API для работы с векторной графикой. Главным ее достоинством является поддержка реализации точного позиционирования, масштабирования и трансформации элементов на Canvas, что обеспечивает гибкую и визуально удобную работу с распознанными документами. Данный подход является необходимым для разработки инструментов выделения, редактирования и удаления сегментов изображения.

Эта система позволила реализовать слой интерактивной графики поверх изображений страниц с возможностью добавления, перемещения и редактирования элементов, которые включают в себя зоны выделения текста, поля ввода, аннотации и другие визуальные маркеры. На ранних этапах реализации позиционирование этих элементов основывалось на абсолютных координатах, которые измерялись в пикселях относительно краёв документа. Такой подход был технически прост, однако показал серьёзные ограничения: при изменении масштаба страницы, разрешения экрана или соотношения сторон изображения происходили заметные искажения — элементы съезжали с исходных позиций, теряя связь с соответствующими участками документа. В результате проведённого анализа и серии экспериментов было принято решение о переходе на относительную систему координат, где все позиции и размеры элементов выражаются не в пикселях, а в процентах от ширины и

высоты страницы. Такой переход стал важным архитектурным шагом, позволившим обеспечить масштабируемость и устойчивость позиционной разметки.

Теперь при любом изменении размеров отображаемого документа, параметров зума или ориентации страницы геометрия элементов сохраняет пропорциональное соответствие исходному изображению. Кроме того, это решение упростило процесс сериализации и синхронизации данных между клиентом и сервером — координаты в относительном формате не зависят от конкретного устройства пользователя и могут быть одинаково корректно интерпретированы при повторном открытии документа. Таким образом, переход от пиксельной к процентной системе позиционирования в связке с Paper.js не только повысил точность и надёжность визуального взаимодействия, но и стал основой для адаптивной и кроссплатформенной архитектуры интерфейса, устойчивой к изменениям масштабирования, разрешения и формата исходных документов.

На уровне пользовательского интерфейса применён гибридный подход: основу составила библиотека Ant Design (antd), а над ней — внутренняя UI-библиотека @quark-uilib/components, реализующая дизайн-систему Гринатома. Это сочетание позволило объединить преимущества готового экосистемного набора компонентов с корпоративными требованиями к стилю, доступности и унификации. Стилизация интерфейсов осуществляется через styled-components, что обеспечивает динамическое применение тем и параметров, а также изолированность CSS в пределах компонентов.

По мере увеличения числа модулей и компонентов возникла необходимость оптимизировать сборку. Изначально проект использовал Webpack, но со временем он стал узким местом. При большом количестве зависимостей и точек входа время запуска дев-сервера достигало 45–50 секунд, а пересборка после правки занимала до 15 секунд. Это снижало скорость итераций и замедляло процесс разработки. После анализа альтернатив команда приняла решение о переходе на Vite, использующий нативные ES-модули и сверхбыструю компиляцию на базе esbuild. В результате время старта проекта сократилось до 3–4 секунд, а Hot Module Replacement стал практически мгновенным — менее секунды даже при сложных сценариях. Помимо скорости, Vite обеспечил простоту конфигурации и нативную поддержку React, TypeScript, CSS Modules и styled-components. Перенос сборки стал ключевым шагом к ускорению разработки и повышению отзывчивости среды.

В проекте активно применяется технология отложенной загрузки (lazy loading) компонентов. Все страницы и крупные модули подгружаются динамически с использованием React.lazy, что позволяет разделять бандл на отдельные чанки и загружать код только при необходимости. Такой подход снижает время начальной загрузки, уменьшает нагрузку на клиент и ускоряет рендеринг интерфейса. В комбинации с React Suspense и fallback-компонентами обеспечивается плавное отображение страниц и повышение отзывчивости приложения, особенно при работе с тяжёлыми редакторами шаблонов, бизнес-логики и графов React Flow. Lazy loading также способствует модульности и масштабируемости: новые страницы можно добавлять без увеличения основного бандла, что упрощает поддержку и развитие системы.

Качество кода поддерживается комплексом инструментов: ESLint и Prettier обеспечивают единый стиль и предупреждают потенциальные ошибки; Husky и lint-staged выполняют автоматические проверки при коммитах, не позволяя невалидному коду попасть в репозиторий.
Для упрощения развертывания была внедрена контейнеризация с использованием Docker. В качестве базового образа для фронтенда использовался Nginx (nginx:alpine), куда копировалась собранная версия приложения из папки dist, а конфигурации Nginx хранились отдельно в директории configs/nginx. Контейнеры запускались с автоматическим перезапуском, а окружение настраивалось через .env и переменные времени. Для локального тестирования и разработки использовались отдельные конфигурации с портами, перенаправленными на 4005, что позволяло одновременно запускать несколько сред.

Для автоматизации сборки и деплоя был настроен CI/CD пайплайн на GitLab, который включал последовательные этапы: установка зависимостей (install), линтинг (lint), тестирование (test), релиз (release), сборка (build), создание Docker-образа (docker) и деплой на серверы (deploy). Пайплайн позволял гибко обрабатывать ветки разработки (dev), тестирования (test) и основную ветку (main), включая возможность ручного запуска этапов для отдельных сценариев, таких как сборка для Росимущество. На этапе сборки использовалась переменная окружения CI_COMMIT_BRANCH для настройки флагов функционала и локализаций, что позволяло динамически включать определённые возможности фронтенда в зависимости от ветки.
Для тестирования используется Jest совместно с React Testing Library, обеспечивающие покрытие логики и компонентов.

Для работы с асинхронными операциями и API-запросами использовались моки через Jest, что позволило тестировать поведение моделей без реальных сетевых вызовов. Проверялись такие сценарии, как установка и изменение атрибутов, добавление и изменение шагов постобработки, сброс всех полей модели, тестирование атрибутов на примере документа, а также управление валидностью полей и блокировкой тестирования.

На уровне компонентов использовалась React Testing Library, которая обеспечивает тестирование взаимодействия с UI и поведение элементов на странице. Это позволяет гарантировать, что пользовательские действия вызывают ожидаемые изменения состояния моделей и корректно отображаются в интерфейсе.

Кроме юнит-тестирования и интеграционного тестирования логики, был внедрён визуальный регрессионный контроль с помощью BackstopJS. Он позволяет автоматически выявлять непреднамеренные изменения во внешнем виде интерфейсов при обновлении зависимостей или рефакторинге компонентов, обеспечивая стабильность UI между версиями. В итоге стратегия тестирования включает полное покрытие всех моделей и сущностей, проверку взаимодействия с API, валидации данных, управление состоянием и визуальную регрессию.

Всё это в совокупности формирует устойчивый, масштабируемый и предсказуемый фронтенд-стек, адаптированный под долгосрочную разработку и сопровождение. Архитектурный переход от модульности к Feature-Sliced Design, выбор React и MobX, внедрение TypeScript и Vite, а также использование специализированных инструментов — React Flow, Paper.js, Quark-UILib — не случайный набор технологий, а результат логически обоснованного системного подхода, направленного на достижение основных немаловажных характеристик системы, таких как гибкость, скорость и надёжность. Опыт, который был получен в процессе создания «Атом.ОКО», демонстрирует, как грамотно построенная архитектура и осознанно сделанный выбор инструментов позволяют выстроить фронтенд, способный не только устойчиво и беспроблемно развиваться на протяжении многих лет, но также быть основой для других корпоративных продуктов Росатома.

Разработка клиентской представляла собой комплексную задачу, которая сочетала в себе продуманный UI, эффективную архитектуру и высокую производительность. В ходе работы удалось построить масштабируемую систему, основанную на React, MobX и Ant Design, где каждая часть

интерфейса — от модуля классификации документов до административной панели — связана единой моделью состояния и системой маршрутизации.

Особое внимание было уделено удобству взаимодействия с документами. Реализованы инструменты для предпросмотра, вращения, перемещения и удаления страниц, а также гибкие фильтры по статусу, дате, пользователю и сорту. Это позволило сделать управление документами интуитивным даже при большом количестве данных.

Механизм пошагового тура с сохранением прогресса в localStorage стал дополнительным элементом UX-улучшения, помогая пользователям быстро освоить интерфейс.

С точки зрения архитектуры и производительности особую роль сыграл переход с Webpack на Vite. Использование нативных ES-модулей, быстрой компиляции через esbuild и стабильного Hot Module Replacement позволило уменьшить время запуска приложения, а обновления после изменения кода стали происходить почти мгновенно. Это повысило скорость разработки и отзывчивость интерфейса даже в сложных сценариях, например при редактировании больших графов React Flow.

Список литературы

1. Мартин Р. Чистая архитектура: искусство разработки программного обеспечения высокого качества. — М.: Вильямс, 2013.
2. Фаулер М. Шаблоны корпоративных приложений / Пер. с англ. — М.: Диалектика, 2004.
3. Никульчев Е., Ильин Д., Гусев А. Модель выбора технологического стека для проектирования программного обеспечения цифровых платформ // Математика. — 2021. — Т. 9, №4. — С. 308.
4. Стефанов, С. React. Быстрый старт / С. Стефанов. – 2-3 изд., стер. – Санкт-Петербург: Питер, 2023. – 304 с.
5. Podila, P. MobX Quick Start Guide / P.Podila, M. Weststrate. – 1st ed. – Packt Publishing, 2018. – 236 с.
6. Ant Design 5.0: сайт URL: https://ant.design/ (дата обращения: 28.07.2025).
7. Vite. The Build Tool for the Web: сайт URL: https://vite.dev/ (дата обращения 28.07.2025)
8. JestJs: сайт URL: https://jestjs.io/ (дата обращения 28.07.2025)

9. React Testing Library: сайт URL: https://testing-library.com/docs/react-testing-library/intro/ (дата обращения 28.07.2025)

10. Эммануил Г. Docker Compose для разработчика // ДМК-Пресс. — 2023. — С. 154–158

11. Feature-Sliced Design: сайт URL: https://feature-sliced.design/ (дата обращения 28.07.2025)

References

1. Martin R. Clean Architecture: A Craftsman's Guide to Software Structure and Design. — Moscow: Williams, 2013.

2. Fowler M. Patterns of Enterprise Application Architecture / Translated from English. — Moscow: Dialektika, 2004.

3. Nikulchev E., Ilyin D., Gusev A. Model for Selecting a Technology Stack for Software Design of Digital Platforms // Mathematics. — 2021. — Vol. 9, No. 4. — P. 308.

4. Stefanov S. React: Quick Start / S. Stefanov. — 2–3rd ed., ster. — Saint Petersburg: Piter, 2023. — 304 p.

5. Podila P., Weststrate M. MobX Quick Start Guide / P. Podila, M. Weststrate. — 1st ed. — Packt Publishing, 2018. — 236 p.

6. Ant Design 5.0: website URL: https://ant.design/ (accessed: 28.07.2025).

7. Vite. The Build Tool for the Web: website URL: https://vite.dev/ (accessed: 28.07.2025).

8. JestJs: website URL: https://jestjs.io/ (accessed: 28.07.2025).

9. React Testing Library: website URL: https://testing-library.com/docs/react-testing-library/intro/ (accessed: 28.05.2025).

10. Emmanuil G. Docker Compose for Developers // DMK-Press. — 2023. — P. 154–158.

11. Feature-Sliced Design: website URL: https://feature-sliced.design/ (accessed: 28.07.2025).